Package Deal

Michael S. Engber, Apple Computer, Inc.

This article discusses some of changes to packages in Newton OS 2.0. Several related topics, such as the details of the package format, Unit Import/Export, and the new Extras Drawer hooks, are mentioned peripherally to point out their relationship to packages and parts. In-depth coverage of those topics is beyond the scope of this article.

To benefit from this article, you should have some experience using the Newton Toolkit (NTK) to write Newton applications for Newtons OS 2.0, and be familiar with the basic concepts of packages and parts. Familiarity with the "Bit Parts" article (see "References and Suggested Reading," at the end of this article) would also be helpful.

Note that some of the details of creating parts with NTK are specific to NTK version 1.6. Earlier versions of NTK may not support parts, and future versions of NTK may support them in a different, more convenient, way.

Basics

Most people think they understand the concepts of packages and parts, but I'm continually hearing nonsensical terms like "auto package," or "my package's RemoveScript." People often confuse packages and parts. Sometimes it's just a slip of the tongue. Sometimes it's indicative of deeper misunderstandings.

A package consists of a collection of zero or more parts. (Yes, it's possible—but pointless—to have a package containing no parts.) A modest example is shown in Figure 1 (in hex).

```
7061636B616765306E6F6E6500000000
0000000100000002000000200000036
00000000000000000000000000000036
000000000058
```

Figure 1: The World's Smallest Package (54 bytes)

Most packages are somewhat larger than 54 bytes, and consist of one or more parts. Table 1 lists the most common kinds of parts.

Type	Frame part?	Description
`form`	Yes	Applications
`book`	Yes	Books
`auto`	Yes	Tricky hacks
`font`	Yes	Additional fonts
`dict`	Yes	Custom dictionaries
`soup`	No	Store w/ read-only soups

Table 1: Part Types

Notice that the second column of Table 1 indicates if a type of part is a frame part. *Frame parts* are basically just a single NewtonScript object — a frame. Conventions for what goes in the frame vary, depending on the type of part. For example, form parts keep their base view's template in a slot named `theForm`.

You can access the actual part frames (of a package's frame parts) via the `parts` slot of the frame returned by `GetPkgRefInfo`. There is bound to be some minor confusion in this regard. Although the argument passed to a form part's InstallScript is commonly called `partFrame`, it is not actually a part frame. In version 1.x it was a clone of the part frame, and in 2.0 it is a frame that protos to the part frame. This detail may change. You should simply rely on the fact that `partFrame.slot` returns the specified slot from the part frame.

Most types of parts are frame parts (store parts being the most notable exception). There are other types of non-frame parts that can be created (e.g., communications drivers), but tools for doing so are not available yet.

Package Objects

In Newton OS 1.x there was no way to directly access packages. The call `GetPackages` returned an array of information about the active packages, but there was no object returned that was itself a package.

In Newton OS 2.0, packages are stored as virtual binary objects (VBOs). A reference to a package VBO is called a package reference (abbreviated pkgRef in many of the related function names). In most ways, package VBOs are the same as regular VBOs created with *`store`*`:NewVBO`. For example, `GetVBOStore` can be used to determine the store on which a package resides.

The downside of representing packages this way is that, like ordinary VBOs, package VBOs are kept in a soup, providing a tempting target for hacking. The details of this soup are private and subject to change. APIs are provided for the supported operations. Anything you do outside of the supported APIs is likely to interfere with the current system (in some subtle way), not to mention breaking in future systems.

Package References vs. Ordinary VBOs

For most purposes, you can think of a package VBO as simply containing the binary data from the package file you created with NTK. There are a few place-holder fields (like the time of loading) that are filled in as the package is loaded, but apart from those, they are byte-wise identical — or so it seems.

If you create a VBO, set its class to `'package`, and then use `BinaryMunger` to copy in the bytes from a "real" package, you'll wind up with something that looks like a package but is still not the real thing. Try passing it to `IsPackage`.

The differences are subtle. One obvious difference is that the real package is read-only. Another, more fundamental, difference is that the fake package is missing the relocation information that is associated with a package during the normal package-loading process. This relocation information is not part of the binary data and cannot be accessed via `ExtractByte`.

The only way to get a real package from the fake one is to pass it to *`store`*`:SuckPackageFromBinary`, which will create a real package from the data. There is no way to convert it in place.

The Extras Drawer

In 1.x, the Extras Drawer was used solely to launch applications. The icons in the Extras Drawer corresponded to form parts and book parts. Package removal was handled by the “Remove Software” button, whose pop-up menu listed packages, not parts.

In Newton OS 2.0, the “Remove Software” button is gone. The Extras Drawer is one-stop shopping for all your package needs. Therefore, the model underlying the Extras Drawer icons is different. For example, a package that contains no form parts still needs an icon to represent it so the user will have something to delete.

The two obvious solutions are to have one icon per package or one icon per part. Neither of these solutions turns out to be satisfactory. Consider the following examples:

- Some multi-part packages require two icons (e.g., form + book part)
- Some multi-part packages require only one icon (e.g., form + auto part)
- Non-frame-parts have nowhere to provide title and icon data (i.e., no part-frame)

The approach adopted by the Extras Drawer is a hybrid of the two, sort of a Miranda rule for packages: “You have the right to an icon. If you cannot afford an icon, one will be appointed for you.”

Any frame part in a package can have an Extras Drawer icon by providing a text slot (and optionally, an icon slot) in its part frame. If no frame part in the package provides an icon, then a generic icon labeled with the package name is created.

This approach preserves the 1.x representation of existing packages, as well as providing flexibility for more complex packages created in the future. The majority of existing packages, which consist of a single form part, get a single icon, just like they always have. Multi-part packages are also handled correctly. The form + book part example gets two icons, and the form + auto part example gets only one icon.

Unfortunately, existing packages that contain no form or book parts get the “court-appointed” representation: a single generic icon (see Figure 2) labeled with their package name. Without more information, there is no way to do much better. Use NTK 1.6 to rebuild these packages and provide a more aesthetically-pleasing title and icon.

Figure 2: Generic Icon

For form parts, the title and icon slots are created from the settings in the Project Settings dialog box (under Output Settings). For other frame parts, you’ll have to create these part-frame slots manually (this is something that could change in future versions of NTK). For example, in one of the project’s text files you would define the title and icon slots with code like the following:

```
SetPartFrameSlot('title, "foo");
SetPartFrameSlot('icon, GetPICTAsBits("foo picture", true));
```

You shouldn’t feel that you need to provide an icon for every part in your package “just because you can.” For most packages, a single icon for the main form part will suffice. Extra icons serve no purpose, and will only confuse the user.

If your package doesn't contain any frame parts (e.g., it contains just a soup part) you can avoid getting the generic icon by adding in a dummy frame part that specifies the title and icon. For example, you might add a form part that displays information about the package.

Active vs. Inactive Packages

Executing a package's InstallScript (and, in the case of form parts, creating the base view) is part of the process of activating a package. Executing the RemoveScript is part of the process of deactivating a package. Prior to Newton OS 2.0, packages were activated when they were installed and deactivated when they were removed. There was no concept of packages existing on a store and being inactive — unless you used a third-party utility. (By inactive package, I mean that the InstallScript has not yet run or, if the package was previously active, the RemoveScript ran during deactivation.)

In Newton OS 2.0, it's possible to have packages visible in the Extras Drawer that are not active. They are identified by either a special snowflake icon, if the user purposely "froze" the package, or by an X'd-out icon, if they are inactive due to an error or because the user suppressed package activation. (See the article "Suppressing and Freezing Packages," in this issue, for further details.)

For the most part, applications need not concern themselves with these details. Any package that correctly handles being installed and removed (by either deletion or card-yanking) should work correctly with user-controlled activation and deactivation.

Invalid References

Invalid references are references to NewtonScript objects residing in a package that is either inactive or unavailable (i.e., on a card that is in the process of being removed). This topic is discussed in the article "Newton Still Needs the Card You Removed" (see "References and Suggested Reading," at the end of this article).

A reference to an object in a package goes bad after the package is deactivated. In 1.x, attempting to access such an object resulted in a -48214 error. In Newton OS 2.0, after a package is deactivated, existing references to it are replaced with a special value. This means that instead of getting a cryptic -48214 error (which also occurs when loading a bad package), you get a more specific error message in the NTK Inspector:

```
!!! Exception: Pointer to deactivated package
```

or

```
<bad pkg ref>
```

or

```
bus error with value 77
```

A related problem occurs for packages on a card that is in the process of being removed. Obviously, the objects in the package are unavailable, and attempting to access them results in the dreaded card-reinsertion dialog. You'll be relieved to hear that the card-reinsertion dialog has not gone away in Newton OS 2.0. However, it has been somewhat improved. It now displays the name of the package that used the invalid reference. There are other, less common, causes for the card-reinsertion dialog appearing (e.g., attempting to access the store) which do not allow for the package to be easily determined. In these cases, the package name is not displayed.

Knowing the name of the offending package is a great help to users trying to figure out why they can't remove a card, but this does little to help a developer figure out why his package is plagued by the "grip of death." There are plans to provide a tool to allow developers to find out what invalid object was accessed.

There is also a new function available for dealing with these problems. Previously, it was impossible to determine if a reference was valid before attempting to access it. In Newton OS 2.0 you can use the function `IsValid` to make this determination.

New Part-Frame Slots

Newton OS 2.0 offers you some new part-frame slots. Some of the slots are data, and some are methods (i.e., they contain functions). You set their values using the `SetPartFrameSlot` function in NTK, as was previously described.

New Part-Frame Data Slots

`app`
`icon`
`title`

In 1.x, the `app`, `icon`, and `title` slots were used to specify a form part's appSymbol and Extras Drawer icon and its Extras Drawer title, respectively. In 2.0, they can be used for any frame part. For non-form-parts, the appSymbol is used to identify the part when using `SetExtrasInfo`.

`labels`

This slot is used to pre-file an Extras Drawer icon. For example, use `'_SetUp` to specify that the icon initially goes in the SetUp folder. See the article "Extra, Extra" (see "References and Suggested Reading," at the end of this article) for a more in-depth example.

New (Optional) Part-Frame Methods

`DoNotInstall`

The `DoNotInstall` message is sent (with no arguments) before the package is installed on the unit. It gives a package a chance to prevent itself from being installed. The message is sent to every frame-part in a package. If any of them return a non-`nil` value, the package is not installed. You should provide the user with some sort of feedback, rather than silently failing to install. For example, a package wanting to ensure it was only installed on the internal store could have a `DoNotInstall` like the following:

```
func()
begin
   if GetStores()[0] <> GetVBOStore(ObjectPkgRef('foo)) then
      begin
         GetRoot():Notify(kNotifyAlert, kAppName, "This package was not installed.
            It can only be installed onto the internal store.");
         true;
      end;
end
```

`DeletionScript`

The `DeletionScript` message is sent (with no arguments) just prior to the package being deleted. This script allows you to distinguish the user scrubbing a package from the user yanking a package's card. A `DeletionScript` typically does "clean up" like deleting soups, deleting local folders, and eliminating preferences from the system soup.

`RemovalApproval`
`ImportDisabled`

These scripts are used to inform a package that some of the units it's importing are being removed. `RemovalApproval` give the package a chance, prior to removal, to warn the user of the consequences of removing the unit. `ImportDisabled` is sent if the unit is subsequently removed. See the "MooUnit" DTS sample code for further details on units.

New Package-Related Functions

`GetPackageNames``(store)`
return value - array of package names (strings)
`store` - store on which the package resides

`GetPackageNames` returns the names of all the packages on the specified store. Note that it returns the names of all the packages, even those that are not active (e.g., frozen packages).

`GetPkgRef``(packageName, store)`
return value - package reference
`packageName` - name of the package
`store` - store on which the package resides

`GetPkgRef` returns a reference to a package given the package's name and the store on which it resides.

`ObjectPkgRef``(object)`
return value - package reference (`nil` if object is not in a package)
`object` - any NewtonScript object

Determines which package an object is in and returns the package reference. Returns `nil` for immediates and other objects in the NewtonScript heap, including soup entries.

A package can get its own package reference by calling `ObjectPkgRef` with any non-immediate literal. For example, `ObjectPkgRef('read`, `ObjectPkgRef("my")`, or `ObjectPkgRef('[l,i,p,s])`. Note that the `InstallScript` of a form part is cloned (by EnsureInternal), and the clone is executed. This means that the above examples wouldn't work because the entire code block — including the argument to `ObjectPkgRef` — resides in the NewtonScript heap. A workaround is to get an object from the package via the argument passed to the `InstallScript` — e.g., `ObjectPkgRef(partFrame.theForm)`. Other types of parts do not have this problem.

`GetPkgRefInfo``(pkgRef)`
return value - info frame (see below)
`pkgRef` - package reference

This function returns a frame of information about the specified package, as shown below.

```
  {
    size:             <int>             // uncompressed package size in bytes,
    store:            <frame>           // store on which package resides
    title:            <string>,         // package name
    version:          <int>,            // version number
    timestamp:        <int>,            // date package was loaded
    creationDate:     <int>,            // date package was created
    dispatchOnly:     <nil or non-nil>, // is package dispatch-only?
    copyprotection:   <nil or non-nil>, // is package copyprotected?

    copyright:     <string>,            // copyright string
    compressed:    <nil or non-nil>,    // is package compressed?
    cmprsdSz:      <int>,               // compressed size of package in bytes

    numparts:      <int>,               // #parts in the packages
    parts:         <part data array>,   // part-frames for the frame parts
    partTypes:     <array of symbols>,  // part types corresponding to data in parts slot

    // other slots are private and undocumented
}
```

IsPackageActive(pkgRef)

return value - `nil` or non-`nil`

`pkgRef` - package reference

`IsPackageActive` determines if the specified package is active or not.

IsPackage(object)

return value - `nil` or non-`nil`

`object` - any NewtonScript object

`IsPackage` determines if the specified object is a package reference.

IsValid(object)

return value - `nil` or non-`nil`

`object` - any NewtonScript object

`IsValid` detects if the specified object is (was) in a package that is no longer active. If the package is on a card in the process of being removed, it will return `nil` and will not cause the card-reinsertion dialog. `IsValid` returns `true` for immediates or objects that do not reside in a package (e.g., in the NS heap or in ROM).

Note that `IsValid` does not deeply check the object.

MarkPackageBusy(pkgRef, appName, reason)

return value - unspecified

`pkgRef` - package reference

`appName` - string describing the entity requiring the package

`reason` - string describing why the package should not be deactivated

`MarkPackageBusy` marks the specified package as busy. This means the user will be warned and given a chance to abort operations that deactivate the package (e.g., removing or moving it). The `appName` and `reason` are used to generate the message shown to the user.

You should mark a package busy if its deactivation will cause you problems. For example, a store part might be providing critical data. Since the user may still proceed with the operation, you should attempt to handle this eventuality as gracefully as possible.

Be sure to release the package as soon as possible so as not to inconvenience the user.

Note that you do not need to use `MarkPackageBusy` on a package because you're importing units from it. Units have their own mechanism for dealing with this problem (`RemovalApproval` et al.).

`MarkPackageNotBusy``(pkgRef)`
return value - unspecified
`pkgRef` - package reference
`MarkPackageNotBusy`t marks the specified package as no longer busy.

`SafeRemovePackage``(pkgRef)`
return value - unspecified
`pkgRef` - package reference
`SafeRemovePackage` removes the specified package. If the package is busy, the user is given a chance to abort the operation.

`SafeMovePackage``(pkgRef, destStore)`
return value - unspecified
`pkgRef` - package reference
`destStore` - store to which to move the package
`SafeMovePackage` moves a the specified package to the specified store. If the package is busy, the user is given a chance to abort the operation; moving a package requires deactivating it, moving it, and then reactivating it.

`SafeFreezePackage``(pkgRef)`
return value - unspecified
`pkgRef` - package reference
`SafeFreezePackage` freezes the specified package. If the package is busy, the user is given a chance to abort the operation.

`ThawPackage``(pkgRef)`
return value - unspecified
`pkgRef` - package reference
`ThawPackage` un-freezes the specified package.

References and Suggested Reading

Engber, Michael S., "Bit Parts." *PIE Developers*, May 1994, pp. 27–29.
This article discusses packages in Newton OS 1.x and creating them with older versions of NTK. Most of the information is still relevant. It is available from the various PIE DTS CDs and archives as well as for ftp from
`ftp.apple.com/pub/engber/newt/articles/BitParts.rtf`.

Engber, Michael S., “MooUnit.” PIE DTS Sample Code, Fall 1995.
This sample code provides documentation on unit import/export and a simple example. It is available from the various PIE DTS CDs and archives.

Engber, Michael S., “Newton Still Needs the Card You Removed.” *Double Tap,* May 1994, pp. 12–18.
This article provides an in-depth discussion of invalid references (to objects in inactive packages). It is available from the various PIE DTS CDs and archives as well as for ftp from

`ftp.apple.com/pub/engber/newt/articles/NewtonStillNeedsTheCard.rtf`.

Goodman, Jerry, “Psychic Archaeology.” 1980, Berkley Books.
This book discusses techniques for researching artifacts of mysterious origin about which very little factual information is known.

Sharp, Maurice, “Extra Extra.” *Newton Technology Journal,* February 1996.
This article discusses various features of the 2.0 Extras Drawer that your package can utilize.